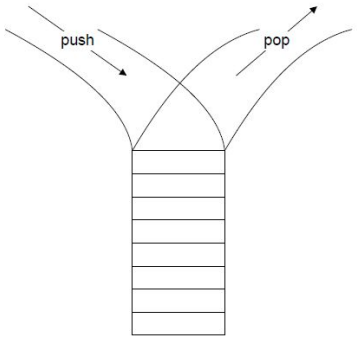
Centrul de pregătire pentru performanță
Colegiul Național „Petru Rareş" Suceava
Data: 12.11.2016
Prof. Narcisa Daniela ŞTEFĂNESCU

STIVA. COADA. LISTA

STIVA

Stiva este o structură de date abstractă. O **stivă** (engleză *stack*) este o structură de date ale cărei elemente sunt considerate a fi puse unul peste altul, astfel încât orice element adăugat se pune în vârful stivei, iar extragerea unui element se poate face numai din vârful acesteia, în ordinea inversă celei în care elementele au fost introduse.

Aşadar, stiva este un caz particular de listă, în care adăugarea sau eliminarea elementelor se face numai în unul din capetele acesteia, iar pentru parcurgerea unei stive implementate pe o structură de tip listă este suficientă referinţa către primul element al listei. În cazul când stiva este implementată sub formă de tablou, punerea şi eliminarea elementelor se face la capătul „din dreapta" (pe poziţia din tablou cu cea mai mare valoare a indicelui). În acest fel, la efectuarea acestor operaţii nu este necesar să se deplaseze celelalte elemente ale tabloului.



Operații

Stiva este concepută pe principiul LIFO ("last in, first out" — ultimul intrat este primul ieşit). Principalele operaţii asupra stivei sunt cele enumerate mai jos.

Creare -Pentru a crea o stiva vidă se iniţializează vârful stivei cu –1 (vârful stivei indică întotdeauna poziţia ultimului element, acestea fiind memorate începând cu poziţia 0).

Inserare - Pentru a insera un element "x" în vârful stivei "S" (operaţia "push") este necesară, în primul rând, verificarea stivei pentru a stabili dacă este sau nu plină. Dacă acest lucru este îndeplinit, se memorează elementul şi se incrementează dimensiunea; în caz contrar sarcina nu se poate îndeplini.

Extragere- Pentru a extrage un element din vârful stivei (operaţia "pop") trebuie ca stiva să nu fie vidă. Dacă nu este, atunci se reţine valoarea din vârful stivei într-o variabilă "x" şi se decrementează vârful.

Vizitare -Accesarea/vizitarea elementului de la vârf stivei presupune determinarea valorii acestuia, valoare care se va reţine într-o variabilă "x", fără a o extrage.

Se poate observa că ultimele trei operaţii au complexitatea O(1), iar prima operație complexitatea O(n).

Cum implementăm o stivă

- Putem implementa o stivă ca un vector în care să reținem elementele stivei. Pentru ca acest vector să funcționeze ca o stivă, singurele operații sunt operațiile caracteristice stivei.

Crearea unei stive vide:

```
#define DimMax 10
typedef int Stiva[DimMax];
Stiva S;
int vf;
vf = -1; //iniţializăm vârful stivei cu -1
```

Inserarea unui element în stivă:

```
if(vf == DimMax +1)
  cout<<"Eroare – stiva este plină \n";
else
  S[++vf] = x;
```

Extragerea unui element din stivă:

```
if(vf < 0)
  cout<<"Eroare – stiva este vidă \n";
else
   x = S[vf --];
```

Vizitarea unui element din stivă:

```
   x = S[vf ];
```

Aplicaţia 1 - PARANTEZE

A.B. are de calculat mai multe expresii la matematică. El a observat că poate scrie dintr-o expresie numai parantezele, fără operanzi şi operatori. O expresie cu paranteze – sau expresie parantezată este orice şir ce conține numai paranteze deschise ‘(‘ sau închise ’)’. Unele expresii cu paranteze sunt construite corect, iar alte expresii sunt scrise incorect, cu paranteze fără corespondent.
Regulile de scriere corectă a unei expresii sunt:

-o expresie începe întotdeauna cu ‘(’

-fiecare paranteză deschisă are o paranteză închisă corespunzătoare, care îi urmează în şir.

De exemplu, expresia (()()) este corectă, iar (()() este incorect construită. A.B. vrea să scrie un program care verifică dacă o expresie cu paranteze este construită corect.

Date de intrare şi de ieşire:
Se citeşte din fişierul ***paranteze.in*** un şir de paranteze şi se cere să se scrie în fişierul ***paranteze.out*** mesajul ‘**DA**’ pentru un expresie corectă şi ‘**NU**’ în caz contrar.
Restricții:

- şirul citit conține cel mult 200.000 de paranteze

Exemplu:

paranteze.in	paranteze.out	Explicaţii
()()(())	DA	-expresia este scrisă corect
(()()()	NU	-expresia (()()() nu este scrisă corect, (nu are corespondent
(()))()(	NU	-expresia este incorectă, (()))()(

Problemă propusă de Prof. Tomşa Gelu (http://olidej.wikispaces.com/)

Algoritmul pentru verificarea corectitudinii unei expresii cu paranteze foloseşte o stivă de caractere, în care se memorează doar parantezele deschise. Paşii algoritmului sunt:
- la citirea unei paranteze deschise, aceasta se adaugă în stivă cu ***PUSH***

- la citirea unei paranteze închise, verificăm dacă stiva conține o paranteză deschisă şi o scoatem din stivă cu ***POP***; dacă apare eroare la operația POP atunci expresia este incorectă (avem o paranteză închisă fără corespondent)

- la sfârşitul şirului, stiva trebuie să fie vidă, k=0, dacă expresia este bine formată

În continuare o să urmărim aplicarea algoritmului pentru şirul bine format **(( ) ( )):**
La început, stiva este vidă:

Şirul citit este **“(( ) ( ))”**
Primul character din şir **(** este adăugat în stivă

Al doilea caracter este tot o paranteză deschisă. Deci în stivă vor fi: ((

Al treilea caracter citit este paranteza închisă pereche a ultimei paranteze adăugate, care va fi extrasă din stivă. Deci va rămâne în stivă: (.

Urmează al patrulea caracter o nouă paranteză deschisă: ((.
Al cincilea caracter este paranteza închisă pereche a ultimei paranteze deschise:), deci în stivă rămâne: (

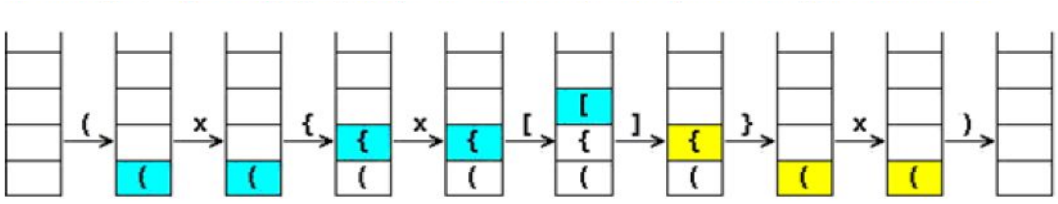
Ultimul caracter este paranteza închisă corespunzătoare primului caracter citit, deci stiva va deveni vidă.

2. Algoritmul de mai sus poate fi adaptat pentru expresii formate cu toate tipurile de paranteze folosite la matematică: paranteze mici, paranteze drepte şi acolade. Condițiile de funcționare a algoritmului sunt aceleaşi, în plus mai trebuie să testăm ca o paranteză închisă să aibă ca pereche în stivă o paranteză de acelaşi tip. Parantezele nu au voie să se intersecteze, de exemplu şirul **([ )]** este incorect format. Spre deosebire de regulile de la matematică într-o expresie cu mai multe tipuri de paranteze este permis ca parantezele drepte sau acoladele să fie incluse între paranteze mici, adică **([]{})** este o expresie corectă.

În acest caz, paşii algoritmului sunt:

- la citirea oricărui tip de paranteză deschisă, aceasta se adaugă în stivă cu ***PUSH***
- orice alt caractere care nu este o paranteză deschisă sau închisă se ignoră
- la citirea unei paranteze închise, verificăm dacă stiva conține o paranteză deschisă de acelaşi tip şi o scoatem din stivă cu ***POP***; dacă apare eroare la operația POP atunci expresia este incorectă (avem o paranteză închisă fără corespondent)
- la sfârşitul şirului, stiva trebuie să fie vidă, k=0, dacă expresia este bine formată

De exemplu, expresia **(x{x[]}x)**este evaluată aşa cum apare în imaginea următoare:



Aplicația 2: conversia unui număr din baza 10 în 2 – folosirea stivei folosind STL

```
#include<iostream>
#include<stack>
using namespace std ;
int main()
{
 stack <int> stiva ;
 int n ;
 cout<<"n=" ;
```

```
cin>>n ;
while (n > 0)
{
  stiva.push( n % 2 ) ;
  n /= 2 ;
}
cout<<"Numarul in baza 2 : " ;
while (!stiva.empty())
{
  cout<<stiva.top() ;
  stiva.pop() ;
}
return 0 ;
}
```

Temă:

Infoarena

http://www.infoarena.ro/problema/paranteze

http://www.infoarena.ro/problema/par

COADA

Coada este o listă în care operaţiile de acces sunt restricţionate la inserarea la un capat şi extragerea de la celălalt capăt. Coada este o structură de date de tip FIFO (First In First Out) în care elementul care este extras este întotdeauna cel care a stat cel mai mult în coadă.

Primul venit primul servit), în care toate inserările se fac la un capăt al ei (numit capul cozii) iar ştergerile (extragerile) (în general orice acces) se fac la celălalt capăt (numit sfârşitul cozii). În cazul cozii, avem nevoie de doi indici, unul către primul element al cozii (capul cozii), iar altul către ultimul său element (sfârşitul cozii). Există şi o variantă de coadă circulară, în care elementele sunt legate în cerc, iar cei doi pointeri, indicând capul şi sfârşitul cozii, sunt undeva pe acest cerc.

Coada este o structură de date abstractă, care poate fi implementată în diferite moduri. Ca şi în cazul stivei, coada poate fi implementată static, reţinând elementele sale într-un vector.
Să considerăm următoarele declaraţii care reprezintă o coadă cu elemente de tip int alocată static:

```
#define DimMax 100    //numărul maxim de elemente din coadă
typedef int Coada[DimMax];
//tipul Coada implementat ca vector
Coada C;              //coada
int Inc, Sf;          //începutul și sfârșitul cozii
```

Elementele cozii sunt memorate în vector de la poziţia *Inc* până la poziţia *Sf*, deci numărul lor este Sf-Inc+1.

int Inc, Sf; //inceputul si sfarsitul cozii

Crearea unei cozi vide:

```
Inc = 0;
Sf = -1;
```

Inserarea unui element în coadă:

```
if (Sf == DimMax-1)        //nu mai avem loc
    cout<<"Eroare\n";
else                       //inserăm elementul x în coada C
    C[++Sf] = x;
```

Extragerea unui element din coadă:

```
if (Inc > Sf)              //coada este vidă
    cout<<"Eroare\n";
else                       //extragem primul element
    x = C[Inc++] ;
```

Accesarea unui element din coadă:

```
X = C[Inc];
```

Aplicaţia 1 - PARANTEZE

Se citeşte de la tastatură o succesiune de paranteze rotunde deschise şi închise până la întâlnirea caracterului ‚.' Întâlnirea unei paranteze deschise determină introducerea acesteia într-o coadă. Întâlnirea unei paranteze închise determină extragerea unui element din coadă. Verificaţi dacă parantezele din şirul citit se închid corect şi determinaţi dimensiunea maximă a cozii.

Soluţie: Vom citi şirul de intrare caracter cu caracter. Dacă este o paranteză deschisă, o inserez în coadă, verific dacă dimensiunea curentă a cozii este mai mare decât dimensiunea maximă, dacă da, reactualizez maximul.

Dacă este paranteză închisă, trebuie să elimin un element din coadă. După citirea tuturor caracterelor, verific dacă în coadă mai sunt paranteze deschise sau nu.

```
#include<iostream>

using namespace std ;
#define DimMaxCoada 100
typedef int Coada[DimMaxCoada];
int Inc=0, Sf=-1, LgMax, Corect = 1;
Coada C;
int main()
```

```
{
   char p;
   do{
     p = cin.get();
     if(p=='(')
     {
        // inserez o paranteza deschisa in coada
        if(Sf == DimMaxCoada -1)
          cout<<"Eroare - coada este plina";
        else
          C[++Sf] = '(';
        if(Sf - Inc +1 >LgMax)
         LgMax = Sf - Inc +1;
     }
     else
     {
        if(p==')')
          if(Inc<=Sf)
            Inc ++;
          else
            Corect = 0;
      }

   } while (p!='.');
   if(Inc <= Sf)
      Corect =0;
   if(Corect == 1)
     cout<<"Parantezele se inchid corect\n";
   else
     cout<<"Parantezele nu se inchid corect\n";

   cout<<"Dimensiunea maxima a
cozii:"<<LgMax;

  return 0 ;
}
```

Aplicaţia 2 – Să se verifice dacă un număr este egal cu oglinditul său.

```
#include<iostream>
#include<queue>
using namespace std ;
int main()
{
  int n, k ;
  queue <int> coada;

  cout<<" n = " ; cin>>n;

  k = n ;
  while (k > 0)
  {
    coada.push(k%10) ;
    k /= 10 ;
  }

  k = 0 ;
  while (!coada.empty())
  {
    k = k * 10 + coada.front() ;
    coada.pop() ;
  }

  if (n == k) cout<<"Numarul este palindrom" ;
  else cout<<"Numarul nu este palindrom" ;

  return 0 ;

}
```

LISTA

Lista este o colecţie, ale cărei elemente pot fi parcurse secvenţial, într-o ordine determinată. Există, deci, un prim element, un ultim element şi o modalitate de a trece de la un element al listei la cel următor. La nivel conceptual, se poate considera că lista este o structură liniară, adică elementele unei liste sunt situate pe o singură linie, la fel cu cele ale unui tablou unidimensional, iar ordinea naturală de parcurgere este cea în care elementele sunt situate în listă. Deosebirea faţă de tablou constă în aceea că, în timp ce numarul de elemente ale tabloului se fixează la crearea acestuia şi nu mai poate fi modificat ulterior, numărul de

elemente dintr-o listă se poate modifica prin adăugări şi eliminări. Tocmai din această cauză considerăm că lista este o colecţie şi nu un tablou.

Adăugarea de elemente la o listă se poate face atât la începutul sau la sfârşitul acesteia, cât şi prin înserarea de noi elemente între cele existente. Eliminarea de elemente se poate face în orice loc al listei.

Elementele unei liste pot fi indexate, la fel ca la un tablou unidimensional, indicele specificând poziţia elementului relativ la inceputul listei. Remarcăm însă că, dacă se inserează sau se elimină elemente, se modifică indicii tuturor elementelor situate în listă în urma lor.

Exemplu Fie lista A B C D Indicii celor patru elemente din listă sunt, respectiv, 0, 1, 2, 3. Prin înserarea elementului X in faţa elementului C, numărul de elemente din listă creşte, iar indicii elementelor C şi D situate după locul înserării creşte cu o unitate: A B X C D Structura de listă se foloseşte foarte frecvent in informatică. Iată numai câteva exemple din domeniul universitar: lista facultaţilor unei universităţi, lista catedrelor unei facultăţi, lista studenţilor dintr-o grupă de studii, lista candidaţilor la un concurs, lista disciplinelor predate la o anumită specializare, lista sălilor de curs ale unei facultăţi etc.

Operații elementare care se pot efectua asupra unei liste:

- crearea unei liste vide;
- inserarea unui element în listă;
- eliminarea unui element din listă;
- accesarea unui element din listă;
- afişarea elementelor unei liste.

Inserarea unui element în listă pe poziţia poz:

```
for(i=n;i>=poz;i--)
   LISTA[i+1]=LISTA[i];
LISTA[poz]=e; n++;
```

Ştergerea unui element în listă pe poziţia poz:

```
e=LISTA[poz];
for(i=poz;i<n;i++)
  LISTA[i]=LISTA[i+1];
n--;
```

DEQUE

Structura de *deque* (pronunțat, de obicei, *deck*) poate fi privită ca o listă cu două capete prin intermediul cărora se şterg sau inserează noi elemente. În literatura de

specialitate aceste capete se numesc *head* şi *tail*, iar deque-ul mai este recunoscut şi ca fiind o coadă cu două capete *(double ended queue)*.

Pentru implementarea unui deque putem recurge la liste dublu înlănțuite sau la un vector static când se cunoaşte numărul elementelor din colecție. Limbajul C++ pune şi el la dispoziția programatorilor o implementare prin intermediul containerului *std::deque* din headerul <*deque*>.

Operații:

Vom putea utiliza această structură de date în situațiile când avem nevoie de următoarele operații (sunt listate cu numele sub care se găsesc în limbajul C++):

▪ *front()*	întoarce primul element
▪ *back()*	întoarce ultimul element
▪ *push_front()*	inserează un element în față
▪ *push_back()*	inserează un element în spate
▪ *pop_front()*	scoate primul element
▪ *pop_back()*	scoate ultimul element
▪ *empty()*	întoarce *true* dacă în deque nu se găseşte niciun element şi *false* în caz contrar

Temă acasă:

Aplicația 1: Vila 2 (.campion, 2005)

Se dă un şir `S` *de* `N` *numere întregi şi* `D` *un număr natural. Se cere să se determine diferența maximă dintre oricare două numere din şir cu proprietatea că diferența în modul a pozițiilor pe care se găsesc în şirul* `S` *nu depăşeşte* `D`*.*
Restricții: `2 ≤ N ≤ 100 000, 1 ≤ D ≤ N/2`*.*

Soluție:
Soluția naivă constă în procesarea tuturor perechilor de numere care se găsesc pe poziții cu diferența în modul mai mică sau egală decât `D`.

```
Algoritmul este:
    ret = 0;
    pentru i = 1, N execută
        pentru j = Max(1, i - D), i execută
            dacă ret < |S[i] - S[j]| atunci
                ret = |S[i] - S[j]|;
        sfârşit_pentru;
    sfârşit_pentru;
    return ret;
Sfârşit.
```

Complexitatea algoritmului este $O(N^2)$, dar pentru limitele problemei este enorm de mare.

Din pseudocodul de mai sus se vede că pentru un indice i fixat, iterăm cu un alt indice j pentru a găsi diferența maximă în modul dintre S[i] şi un alt număr din intervalul [i - D, i]. Însă, diferența dintre cel mai mare număr şi cel mai mic număr este cel puțin la fel de bună ca rezultatul de la j-ul anterior. Astfel, pentru indicele i fixat succesiv cu 1, 2, .., N, considerăm secvența [i - D, i] în care determinăm valoarea maximă şi valoarea minimă, iar diferența lor o comparăm cu rezultatul cel mai bun obținut până în acel moment.

Pentru fixarea ideii, să urmărim cum putem determina eficient valoarea maximă din fiecare secvență [i - D, i].

Observație: Fie i_1, i_2 doi indici astfel încât $i - D \le i_1 < i_2 \le i$.

1. Dacă $S[i_1] \le S[i_2]$ atunci, cât timp cei doi indici vor fi în intervale de tipul [i - D, i], valoarea de pe poziția i_2 va fi întotdeauna preferată valorii de pe poziția i_1. Când se ajunge la un interval care nu-l mai conține pe i_1, i_2 rămâne în continuare preferat.
2. Dacă $S[i_1] > S[i_2]$ atunci şi valoarea de pe poziția i_1 şi cea de pe poziția i_2 sunt candidate la maxim, la momentul curent sau în viitor.

Cu această observație deducem că într-o secvență [i - D, i] vom avea un şir strict descrescător de numere: $T_i = S[i_1] > S[i_2] > ... > S[i_K]$, unde $S[i_1]$ elimină toate elementele S[j], cu $S[j] \le S[i_1]$ şi $i - D \le j < i_1$, $S[i_2]$ elimină toate elementele S[j], cu $S[j] \le S[i_2]$ şi $i_1 < j < i_2$, $S[i_3]$ elimină toate elementele S[j], cu $S[j] \le S[i_3]$ şi $i_2 < j < i_3$ ş.a.m.d. De reținut este că niciuna dintre valorile de pe pozițiile eliminate nu poate fi maximă, motiv pentru care le-am putut elimina. Şirul T_i are următoarele proprietăți:

- se termină pe poziția curentă, adică are loc egalitatea $i_K = i$ întrucât poziția i nu va fi eliminată de nicio altă poziție;
- valoarea căutată, adică maximul dintre numerele din secvență, se găseşte pe prima poziție.

Când vom avansa la secvența următoare, [i - D + 1, i + 1], vom forma şirul T_{i+1} ştergând din indicii i_1, i_2 ... atâta timp cât nu se găsesc în intervalul curent şi vom şterge din pozițiile i_K, i_{K-1} ... cât timp $S[i + 1] \ge S[i_K]$, $S[i + 1] \ge S[i_{K-1}]$ ...

Şirul T_i poate fi păstrat prin intermediul şirului de indici $i_1 < i_2 < ... < i_K$. Operațiile pe acest şir se efectuează doar pe la cele două capete, aşadar poate fi implementat cu ajutorul unui deque.

Pentru S[] = {5, 9, 4, 7, 4, 1} şi D = 3 obținem următoarele stări ale unui deque:

- $\langle \widehat{5\ [1]} \rangle$;
- $\langle 5\ [1] \rangle \Leftarrow 9\ [2]$ de unde se obține $\langle \widehat{9\ [2]} \rangle$;

- $\langle 9\ [2]\rangle \Leftarrow 4\ [3]$ de unde se obține $\langle \widehat{9\ [2]},\ 4\ [3]\rangle$;
- $\langle 9\ [2],\ 4\ [3]\rangle \Leftarrow 7\ [4]$ de unde se obține $\langle \widehat{9\ [2]},\ 7\ [4]\rangle$;
- $9\ [2] \Leftarrow \langle 7\ [4]\rangle \Leftarrow 4\ [5]$ de unde se obține $\langle \widehat{7\ [4]},\ 4\ [5]\rangle$;
- $\langle 7\ [4],\ 4\ [5]\rangle \Leftarrow 1\ [6]$ de unde se obține $\langle \widehat{7\ [4]},\ 4\ [5],\ 1\ [6]\rangle$;

Cum fiecare indice din 1, 2, ..., N este adăugat şi şters cel mult o dată din deque, complexitatea finală este O(N).

Bibliografie:

Ema Cerchez – Programarea in limbajul C/C++
Marius Stroe – Articol Deque
Dan Praxiu – Aplicatii STL